

# Opportunistic Evolution: Efficient Evolutionary Computation on Large-Scale Computational Grids

Keith Sullivan and Sean Luke  
Department of Computer Science  
George Mason University  
4400 University Drive MSN 4A5  
Fairfax, VA 22030 USA  
{ksulliv2, sean}@cs.gmu.edu

Curt Larock, Sean Cier, and Steven Armentrout  
Parabon Computation, Inc.  
11260 Roger Bacon Dr., Suite 406  
Reston, VA 20190 USA  
{clarock, sceir,  
sarmentrout}@parabon.com

## ABSTRACT

We examine opportunistic evolution, a variation of master-slave distributed evaluation designed for deployment of evolutionary computation to very large grid computing architectures with limited communications, severe evaluation overhead, and wide variance in evaluation node speed. In opportunistic evolution, slaves receive some  $N$  individuals at a time, evaluate them, and then run those individuals through their own mini evolutionary loop until some fixed wall clock time has been exceeded. Our implementation of opportunistic evolution may be used in conjunction with either a generational or, for maximum throughput, an asynchronous steady-state evolutionary model in the master. Opportunistic evolution is strongly exploitative. We perform initial experiments comparing the technique with a traditional master/slave model, and suggest possible classes of problems for which it might be apropos.

## Categories and Subject Descriptors

G.1.6 [Optimization]: Global Optimization; C.2.4 [Distributed Systems]: Distributed Applications

## General Terms

Algorithms

## Keywords

Distributed Evolutionary Computation

## 1. INTRODUCTION

As part of a NASA-funded project, we have developed *Origin* [12], a deployment of the ECJ evolutionary computation toolkit to a commercial Java-based grid computing platform known as *Frontier*, by Parabon Computation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

Copyright 2008 ACM 978-1-60558-130-9/08/07 ...\$5.00.

Frontier enables grids of very large numbers (tens or hundreds of thousands) of nodes—typically PCs with unused CPU cycles in large organizations. In addition to its computational grid deployment platform, Frontier also acts as a brokerage system: Parabon contracts with large organizations to host Frontier on their personal computers, then sells access to these machines to Parabon's clients.

This model imposes considerable security constraints, as organizations are understandably concerned about compromises to their infrastructure. These security constraints in turn place restrictions on communications between nodes and on the speed of communications (due to firewalls and virtual private networks). Additionally, available nodes may be anywhere in the world, resulting in significant additional communications costs. Finally, there may be a wide variance among the speeds of the nodes available.

How might one host a distributed evolutionary computation system on such a platform? One could use Frontier's remote machines as slave nodes in a distributed (master-slave) evaluation system. However, the severe communication overhead can hinder this approach unless evaluations are very long. Ordinarily we would turn to an island model, which can operate in a low-communication environment. However, the remote machines are usually not allowed to communicate with one another: only with the master node.

Our solution to this conundrum is, ironically, a hybrid of the island and distributed evaluation models, in combination with either generational or asynchronous steady-state evolution, which we collectively term *opportunistic evolution* (OE). Here, a master evolutionary computation process sends groups of individuals to each remote slave process to be evaluated. But instead of evaluating them and returning them immediately, each slave engages in its own mini evolutionary loop for a period of time sufficient to justify the communication costs. The slave then returns individuals to the master. This curious configuration is motivated by the desire to use constrained Frontier to its maximum capacity. We do not argue for some evolutionary advantage inherent in OE: only that it makes good use of constrained but plentiful computational resources. The question we ask is: for what problems will it perform well?

In this paper we first discuss parallel methods, ECJ, and Frontier, then introduce the technique and report our experiments with it. On Origin, we primarily use an asynchronous steady-state evolution model as it makes better

use of nodes of widely varying speed. But this paper we will instead focus on a generational model, as it is easier to directly compare OE with a plain distributed evaluation procedure in order to more easily assess what kinds of problems may be amenable to the technique.

The rest of this paper is organized as follows: Section 2 discusses parallel evolutionary methods including ECJ. Section 3 introduces Frontier and Origin, while Section 4 introduces OE. Section 5 discusses experimental results, and Section 6 provides conclusions.

## 2. PARALLEL EVOLUTIONARY COMPUTATION METHODS

Because of its population-oriented structure, evolutionary computation has long been an easy target for parallel computing (see for example, the surveys in [4] and [11]). Much of the parallel evolutionary computation literature may be broken, perhaps unfairly, into roughly four methods:

### *Distributed “Master/Slave” Generational Evaluation.*

When a generational evolutionary computation process needs to evaluate individuals, it does so by sending them to remote “slave” processes to be evaluated in parallel. The slaves then return the individuals or their assessed fitness values. This may be done synchronously (for generational evolution) or asynchronously (for steady-state evolution).

*Directed Breeding and Evaluation.* The population is broken into multiple subpopulations, each subpopulation residing on a different slave process. The master evolutionary computation process maintains an overall map of the population and its respective fitness values. When the evolutionary computation process wishes to breed individuals, it does so by directing multiple remote “slave” processes to exchange the individuals and cross them over or mutate them. The slaves then may assess the fitness values of the new children, sending fitness information back to the master so it may update its central map.

*Island Models (or “Demes”).* The population is again broken into multiple subpopulations, each subpopulation residing on a different process. However, there is no master per se. Instead, each process maintains its own independent evolutionary loop, selecting, breeding, and evaluating individuals from its own subpopulation. Occasionally copies of individuals from one subpopulation will “migrate” to another subpopulation over the network. Island models may operate at various levels of synchrony: for example, after an island receives migrants, it may wait to incorporate them into its population at its leisure (or decide not to).

*Fine-Grained (or “Cellular”) Models.* Here the population is distributed among many processes (often one processor per individual) which collectively take part in a shared evolutionary process. Each process is responsible for updating its individuals through selecting individuals from other processes, performing crossover and mutation, and evaluating the resulting children. Processes are often, but not always, organized spatially and selection prefers individuals of neighboring processes.

Some of these techniques are orthogonal and may be composed in various ways: for example, each process in an island model may act as a master in a distributed evaluation scheme, with its own set of evaluation slaves.

Because we are concerned with communications overhead, it is important to note the communications characteristics typical of these methods. Fine-grained models generally assume a very high degree of communications fabric, such as a vector processor system or shared memory system, and so are not as relevant as the other methods. Distributed evaluation and directed breeding/evaluation techniques typically require transfer of all individuals in the population to and from various slaves, either to form a new population, or to evaluate its individuals. Such an approach is thus appropriate when evaluation or breeding costs are sufficiently large to justify the communications overhead (see [5], p. 36). On the contrary, island models are adept at utilizing a large gamut of communication speed and bandwidth: it simply determines the rate at which individuals may migrate throughout the joint population. At the extreme, with no communication, island models simply approximate independent evolutionary runs. Last, and importantly, all of the above methods, except for distributed evaluation, presume that the slaves can communicate with one another. There are ways to work around this: for example, island models may transfer individuals to one another by communicating solely through a central “master” acting as an intermediary: but this can be very costly depending on the network fabric.

One particular variation bears relationships with our technique. So-called “memetic” algorithms augment the evolutionary loop by performing local optimization on individuals during their fitness evaluation. Memetic algorithms have been used with synchronous and asynchronous island models [2, 14, 3] and with directed breeding/evaluation [6]. But since the additional cost of a memetic algorithm lies in its easily distributable local search procedure during an individual’s evaluation, performing distributed evaluation would seem a natural fit. If this local search took the form of hill-climbing, a memetic algorithm would approximate an OE technique with a remote population of size one or two.

### 2.1 ECJ and Distributed Evaluation

ECJ [9] is a Java-based evolutionary computation toolkit developed at George Mason University, and provides for various approaches to parallelization: island models, multiple threads for evaluation or breeding, and distributed evaluation over many remote processors. It is this third mechanism which is the focus of this paper.

*Generational Distributed Evaluation.* In the distributed (or “master-slave”) evaluation model, a central evolutionary computation process (the master) farms out evaluations of individuals to parallel slave processes. This is a straightforward method for handling parallel generational evolutionary runs, and is Algorithm A in Grefenstette’s 1981 report [7].

A naive approach to distributed evaluation simply cuts the population into  $N$  sections and sends each section to a different slave. ECJ adopts a somewhat more flexible procedure: the population is broken into *groups* and each group is submitted to a queue. When a slave is available, it requests the next group from the queue and begins processing the individuals in the group. When a slave has completed evaluating all the individuals in a group, it is assigned a

new group from the queue if there are any left. If a slave goes offline, its currently-assigned group is placed back in the queue. New slaves may come online at any time: they are simply given the next group to perform.

**Asynchronous Evolution: Steady-State Distributed Evaluation.** Here, when a slave comes available, individuals are bred or created and submitted as a group to the slave. When a slave has completed its evaluation, the returned individuals are inserted into the population, displacing some existing individuals. This model is not often implemented, but has been described in the past [11] and was one of the original four algorithms (Algorithm B) in [7]. Asynchronous evolution tolerates very wide variance in evaluation time: if a group, for whatever reason, takes a long time to process, it simply doesn't get to participate in the evolutionary cycle until it is finished.

Asynchronous evolution in ECJ has two stages: population initialization and population steady-state. During initialization individuals are created at random and handed (as groups) to the slaves; and when slaves return individuals, they are added to the population until it has grown to full size. Once the population has reached full size, the system switches to steady-state mode. Now slaves receive groups of individuals newly bred from the population; and an individual completed by a slave is added to the population, replacing an existing individual.

### 3. FRONTIER AND ORIGIN

The Frontier<sup>®</sup> Grid Platform is a commercial grid computing platform developed by Parabon Computation, Inc. It enables applications launched from an ordinary desktop-class computer to harness the idle computational capacity of thousands of computers. The platform is comprised of three main components: The *Frontier Compute Engine* is a desktop application that utilizes the idle computational power of an Internet-connected machine to process small units of computational work called tasks. The *Client Application*, executed from a single computer, is a domain specific application configured to execute compute-intensive jobs on many compute engines. The *Frontier Server* is the central hub of the Frontier platform. It communicates with the client application and multiple compute engines. It coordinates the scheduling and distribution of tasks; maintains records identifying all compute engines, client sessions, and tasks; and ensures the platform's consistency and reliability. Communication with the client application occurs within the context of a session, during which jobs can be launched, monitored, and terminated.

**Jobs and Tasks.** Computational work to be performed on Frontier is grouped into a single, relatively isolated unit called a job. Within a job, work is divided into an arbitrary set of individual tasks, with each task being executed independently on a single compute engine. Tasks running on Frontier have the following characteristics:

- Tasks cannot communicate with other running tasks.
- All communication to a task takes place at the instantiation of the task.
- All communication from a task (e.g., reporting results) occurs in the form of periodic and final status reports.

Frontier jobs must be divided into a set of relatively small tasks, each of which is independently executed on an engine before reporting its final results. The results of these tasks are then gathered by the client application and assembled like pieces of a puzzle to form a coherent whole. This requires that the amount of work a task performs be small enough both to be processed effectively given the resources (i.e., memory, disk storage, etc.) available to a compute engine and to return a final result within a relatively short time—generally, a few minutes to a few hours.

Further, as Internet communication is inherently high-latency, the time required for a round trip between tasks running on two different nodes can be quite long—often on the order of several seconds and possibly as long as minutes. Thus, frequent communication between tasks is not feasible. Though inter-task communication may be more feasible in the future, this functionality is not supported in the current version of Frontier.

**High Compute-to-Data Ratio.** The individual machines that provide Frontier's computational power may have modest bandwidth connections to the Frontier server. Further, the central server must communicate with many of these nodes simultaneously, meaning that communications bandwidth on the server side is at a premium as well. This means that sending large amounts of data to nodes and returning large results can take significant amounts of time.

However, after a task's data has been sent to a node and before its results are sent back, the compute engine can efficiently crunch away on a task for minutes or hours. Thus, tasks run most efficiently if they have large amounts of computation to perform and relatively small pieces of data required and results to report. This is known as a high compute-to-data ratio.

A task with a very high compute-to-data ratio is compute-limited and tends to scale and run as well on Frontier as on a traditional cluster of machines. On the other hand, a task with a very low compute-to-data ratio is bandwidth-limited and spends most of its time transferring data back and forth to and from the server. A job comprised of entirely such tasks could, take longer to complete on Frontier than on a single machine.

**Origin.** The Origin<sup>™</sup> Evolutionary SDK (software development kit), also developed by Parabon, enables ECJ applications to run on Frontier with little or no modification. Together, ECJ, Origin and Frontier provide a convenient means for performing evolutionary computation on large-scale computational grids.

### 4. OPPORTUNISTIC EVOLUTION

Our approach is motivated by the high communications costs involved in massive distributed grids, and in overcoming certain constraints placed on those grids to enable them to run sufficiently securely. In a grid computing application using the background CPU cycles of PCs in large organizations, communication between processes on different machines is generally forbidden given the security concerns of running jobs on unprotected machines behind organizational firewalls. Furthermore, communication between processes in different organizations may be impossible due to firewall protections for each organization. These constraints on inter-

process communication generally rule out island models and related parallel techniques. Instead two other options are intuitive: running an independent evolutionary computation job on each process; and using processes to distribute the evaluation tasks, breeding tasks, etc.

We have chosen to distribute evaluation tasks on the grid. The primary difficulty in doing so is the communication and setup costs. There is a significant overhead involved in shuttling individuals to the remote slave, constructing the evaluation environment, and returning either the individuals or their fitness values after evaluation. There is also an absolute bandwidth cap on the number of individuals that the master can send to the slaves over the network fabric.

If an evaluation process were sufficiently long, it might justify the cost of shuttling the individual to the remote slave, processing it, and returning the individual (or its fitness) back to the master. But how long is long? For a large distributed grid spread over the internet, communications costs are surprisingly high: we have found that evaluations might need to run for as much as a minute to make effective use of the system. On a more local distributed environment such as a beowulf cluster, these costs are much lower, but they are still significant. Thus while optimization tasks involving “slow” evaluations (multiagent models, robotics simulations, etc.) are good targets for distributed evaluation, “fast” evaluations cannot take advantage of the distributed resources sufficiently efficiently.

Instead of performing longer evaluations, we have instead chosen to perform more of them. Opportunistic evolution augments our distributed evaluation (asynchronous steady state or generational) by performing mini-evolution processes on the remote slaves. We first construct a slave with a maximum wall-clock time more than sufficient to justify the setup costs of the slave. We then send to the remote slave a job with as many individuals as is reasonable. The slave then evaluates its job. If a slave’s wall-clock time has been exceeded, it immediately returns its job. Otherwise, treating the evaluated individuals as an initial population, the slave enters into its own evolutionary loop, selecting parents, breeding new children, evaluating the children, and reintroducing them to the population. When the wall clock time has been exceeded, the current population is returned in lieu of the original individuals. The evolutionary loop details are up to the experimenter.

We know of one paper from the literature which has implemented opportunistic evolution in the past [1], and although it is an early and important one, it is typically mis-cited as describing ordinary Master/Slave evolution. In that paper, the number of mini-“generations” on each slave was fixed (to 10), whereas in our case the number varies, being restricted by wall-clock time.

## 5. PRELIMINARY EXPERIMENTS

We have begun experimenting with OE. To keep things simple, our initial experiments have compared the performance of generational OE against generational “master-slave” distributed evaluation; we did not apply asynchronous evolution so as to eliminate the uncontrolled factor of variance in evaluation time, particularly with regard to genetic programming problems. We also have chosen to run OE in a controlled environment, namely a compute cluster with a fixed number of nodes.

We compared OE and master-slave evaluation by running each for 5 minutes of wall-clock time and comparing performance results. OE slaves continued to perform evolution for a fixed amount of time: thus OE’s total number of generations on the “master” was expected to be much lower than distributed master-slave evaluation. We chose two standard genetic programming problems and two vector problems common to genetic algorithms or evolution strategies. Our two genetic programming problems are the 10-bit Even Parity and Artificial Ant problems. A solution to the 10-bit Even Parity problem discovers a genetic programming tree function of 10 inputs which outputs whether or not there is an even number of 1-bits among the inputs. A solution to the Artificial Ant problem is a GP tree which, when executed, directs an ant to eat as many pellets as possible in a grid world. We used the well-known Santa Fe trail, which contains 89 pieces of food. The formal description of both problems may be found in [8]. Genetic programming experiments employed 50 compute nodes.

The vector problems are Rastrigin and Hierarchical If-And-Only-If. The Rastrigin problem tries to minimize the function  $f(x) = 10n + \sum_{i=1}^n (x_i^2 - 10 \cos(2\pi x_i))$  using a fixed-length vector of  $n$  real values, each  $-5.12 \leq x_i \leq 5.12$ . The Rastrigin function has many local minima, and the global optima is at the origin with a value  $f(x) = 0$ . For our experiments, we chose  $n = 500$ , and used 30 compute nodes.

The Hierarchical If-And-Only-If (H-IFF) problem [16] has multiple local optima and two global optima. Using a bit representation, H-IFF groups bits into blocks, then combines these blocks into larger blocks, forming a hierarchy of blocks. Assuming  $n = 2^k$ , as you move up the hierarchy, the number of blocks halve and the size of the blocks double. The H-IFF problems tries to maximize the function:

$$f(B) = \begin{cases} 1 & |B| = 1 \\ |B| + f(B_L) + f(B_R) & \forall i, b_i = 1 \text{ or } \forall i, b_i = 0 \\ f(B_L) + f(B_R) & \text{otherwise} \end{cases}$$

where  $B$  is a block of bits,  $\{b_1, b_2, \dots, b_n\}$ ,  $|B|$  is the size of the block, and  $B_L$  and  $B_R$  are the left and right halves of  $B$ . From this fitness function, its clear that a string of all ones or all zeros has maximum fitness. Minimum fitness occurs with a string of alternating zeros and ones. This formulation generates blocks which are not separable, unlike similar problems e.g, Royal Roads [10]. For our experiments, we chose  $n = 2^{10} = 1024$ , and used 20 compute nodes.

Experiments consisted of 50 independent runs. The 10-bit parity problem used a population of 5000, the Artificial Ant problem used a population size of 1000, while the Rastrigin function used a population size of 3000 and the H-IFF problem used 2000. All problems used a chunk-size of 100. In the Artificial Ant problem, the ant is limited to 400 moves. For all problems, OE was given a one second slave-evolution wall clock time limit, which resulted in approximately 16 generations on each slave for 10-Bit Parity, approximately 124 generations on Artificial Ant, approximately 214 generations on H-IFF, and approximately 150 generations on Rastrigin. Due to the varying number of generations completed per run, in all the figures we report maximal best-so-far curves, i.e, for short runs, we pad the data to match the largest number of generations completed for that problem.

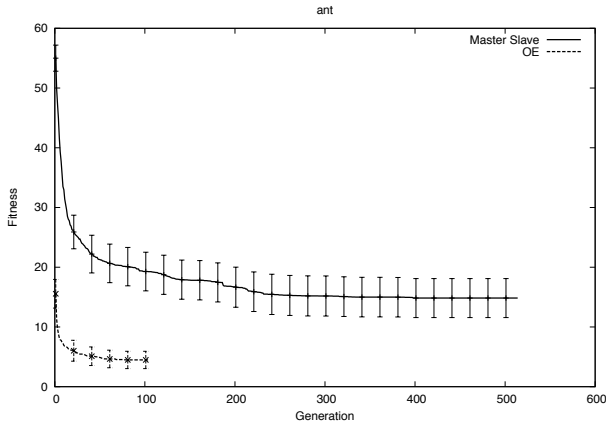


Figure 1: Best-so-far curves for the Artificial Ant experiments. Lower fitness is better.

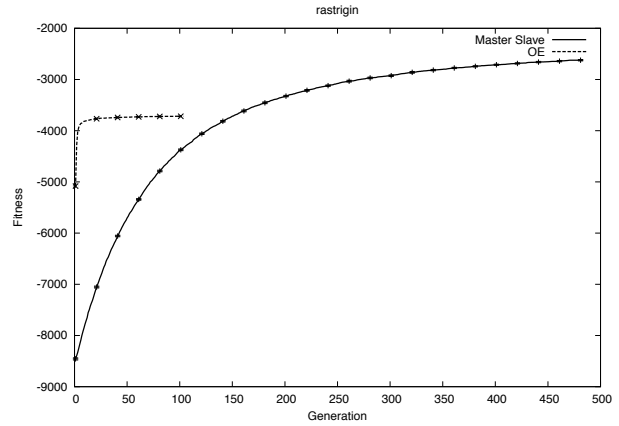


Figure 3: Best-so-far curves for the Rastrigin experiments. Higher fitness is better.

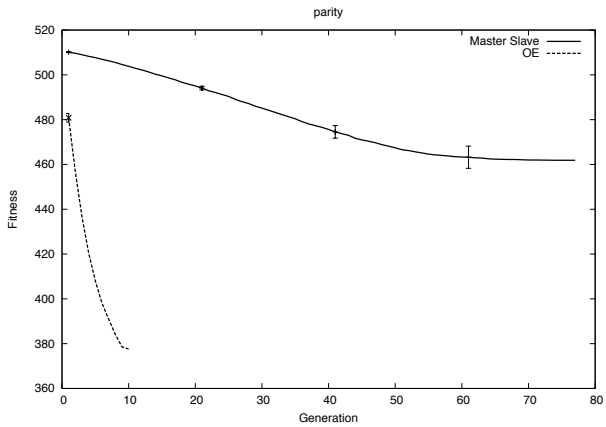


Figure 2: Best-so-far curves for the 10-Bit Parity experiments. Lower fitness is better.

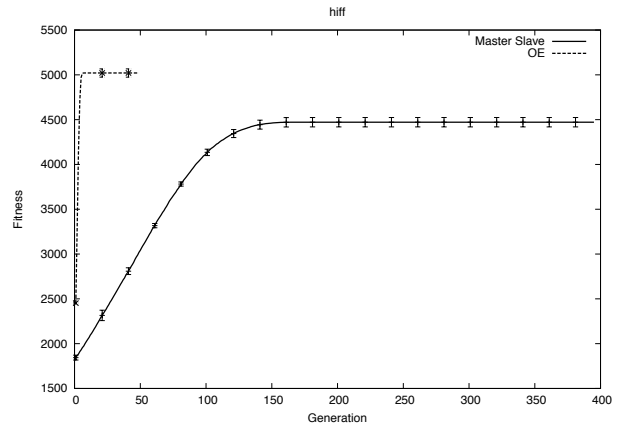


Figure 4: Best-so-far curves for the H-IFF experiments. Higher fitness is better.

**Results and Discussion.** Figure 1 and Figure 2 show the results on the Artificial Ant and 10-Bit Parity problems respectively. Note that in both problems, individuals change size during evolution. In both problems OE outperforms traditional master-slave by a wide margin. This performance was not consistent, however. Figure 3 shows the best-so-far curve for the Rastrigin problem, where OE performs worse than traditional master slave.

The primary advantage of OE is that it is able to stuff in many more evaluations in the same period of time, because it is less reliant on network lag.<sup>1</sup> The disadvantage is the OE is aggressively modifying individuals with a form of semi-local search on the slaves, resulting in much more exploitation than exploration.<sup>2</sup> For some problems, such as artificial ant, either higher exploitation is desirable, or (more likely) the significant increase in number evaluations and lower lag is worth the cost in additional exploitation.

For what kinds of problems might OE be a good

pick? We think it may not simply be an issue of exploration/exploitation tradeoff: in informal experiments we increased the mutation rate on the slaves, but the results remained the same. Rather, recent work has shown that breaking a population into smaller evolutionary chunks may be evolutionarily advantageous to so-called *compositional problems* [15], where full solutions consist of subsolutions which may be discovered independently, at least in part. Compositional problems have been shown to be specifically amenable to island models [13], and are also a natural fit for coevolutionary techniques. Likewise, as OE is essentially setting up small temporary islands, we believe that compositional problems might be helpful here as well.

To test this, we compared OE with master-slave using the H-IFF function, a strongly compositional problem. Figure 4 shows the best-so-far curves for the H-IFF problems: here, OE wins readily.

<sup>1</sup>This advantage is lessened when individuals are sufficiently large, since OE must both send *and* return individuals over the network, while master-slave need only send individuals and return fitnesses.

<sup>2</sup>This is same basic disadvantage may exist for memetic algorithms.

## 6. CONCLUSIONS AND FUTURE WORK

This paper introduced opportunistic evolution (OE), a new parallel evolutionary algorithm designed for large scale grid computing systems. OE distributes groups of individuals to remote slave nodes to be evaluated, and given time it performs small evolutionary loops on those slave nodes before returning them to the master. OE is designed to be robust in the face of high communications and initialization costs and wide variance in evaluation resources found in such systems. OE's development was motivated by the demands of Origin, a hosting of the ECJ evolutionary computation toolkit on the Frontier massive grid computing framework. We identified advantages and disadvantages of OE, and discovered examples of OE outperforming traditional master-slave models (and vice versa).

Our initial results suggest avenues for future work. We plan to examine more problem domains to better understand the types of problems on which OE will perform well, and how OE compares to island models. In addition, we plan to examine different evolutionary algorithms on the slaves and the master, and the effects of various parameters (mutation rate, number of individuals sent to the slaves, etc.). Last, we are interested in the scalability of OE, especially using genetic programming to evolve support vector machines.

## 7. REFERENCES

- [1] R. Bianchini and C. Brown. Parallel genetic algorithms on distributed-memory architectures. Revised Version 436, Computer Science Department, University of Rochester, Rochester, NY 14627, 1993.
- [2] R. Bradwell and K. Brown. Parallel asynchronous memetic algorithms. In E. Cantú-Paz and B. Punch, editors, *Evolutionary Computation and Parallel Processing*, pages 157–159, 1999.
- [3] A. E. Calaor, A. Y. Hermosilla, and B. O. C. Jr. Parallel hybrid adventures with simulated annealing and genetic algorithms. In *Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks*, 2002.
- [4] E. Cantú-Paz. A survey of parallel genetic algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 S. Mathews Ave., Urbana, IL 61801, 1997.
- [5] E. Cantú-Paz. *Designing Efficient and Accurate Parallel Genetic Algorithms*. PhD thesis, University of Illinois, 1999.
- [6] J. Digilakis and K. Margaritis. Performance comparison of memetic algorithms. *Journal of Applied Mathematics and Computation*, 158:237–252, October 2004.
- [7] J. Grefenstette. Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Computer Science Department, Vanderbilt University, P.o. Box 1679, Station B, Nashville, TN 37235, 1981.
- [8] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.
- [9] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, J. Harrison, J. Bassett, R. Hubley, and A. Chircop. ECJ: A Java-based evolutionary computation research system. <http://cs.gmu.edu/~eclab/projects/ecj/>, 2007.
- [10] M. Mitchell, S. Forest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the First European Conference on Artificial Life*, 1992.
- [11] M. Nowostawski and R. Poli. Parallel genetic algorithm taxonomy. In *Knowledge-Based Intelligent Information Engineering Systems*, pages 88–92, 1999.
- [12] ORIGIN: Evolutionary SDK. <http://www.parabon.com/developers/origin.jsp>, 2008.
- [13] Z. M. Skolicki. *An Analysis of Island Models in Evolutionary Computation*. PhD thesis, George Mason University, 2007.
- [14] J. Tang, M. H. Lim, Y. S. Ong, and M. J. Er. Parallel memetic algorithm with selective local search for large scale quadratic assignment problems. *International Journal of Innovative Computing, Information and Control*, 2(6):1399–1416, 2006.
- [15] R. A. Watson. *Composition Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis*. PhD thesis, Brandeis University, 2002.
- [16] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In *Proceedings of Parallel Problem Solving from Nature*, 1998.